Embeddings/Attention/Transformers

Thibault Randrianarisoa

University of Toronto, Winter 2026

March 17, 2026

Overview

- Recap of NLP Tasks
- Intro to Embeddings
- Intro to Attention
- Goal is to catch you up on LLMs

NLP Tasks

- Classification (Documents, Spans, Tokens)
  - Hate speech detection
  - Spam filtering
- Generation (Question Answering, Summarization, Free-text generation, . . .)
  - Translating natural language into SQL queries
  - “Hey siri what is the weather like?”
  - Chatbots
  - Talk to a transformer

NLP Tasks

- Regression (Essay scoring, like count prediction)
  - How many retweets will this tweet get?
- Information Extraction
  - Who are the people mentioned in this text?
  - What date was the procedure performed?
- Document Retrieval
  - Google search
  - Automatic literature review

NLP concepts

Some names we will be using throughout this lecture:

- **Token** - A single "atom" of text, usually a word
- **Document** - A complete datapoint of text
- **Span** - A subset of a document, a group of Tokens
- **Vocabulary** - A list of all possible tokens

Spam

We begin by revisiting a familiar example from earlier lectures: determining whether an email is spam or not.

- What kind of task is it ? → classification
- How would you approach it?

More formally: Consider a set of observations $(x_i, y_i)_{i=1:N}$ where $y_i = 1$ means datapoint i was spam, and x_i is the text of the email.

Our goal is to create / learn a function f_θ such that $f_\theta(x_i) = p(y_i|x_i)$

Heuristics

Perhaps it is possible to find some heuristics that work:

- If x_i contains any prescription medication name $\hat{y}_i = 1$
- If x_i is mostly capital letters $\hat{y}_i = 1$
- If x_i contains 'Make **Amount** every **Time Period**" $\hat{y}_i = 1$

Baseline for more advanced M2 approaches

```
import re

SPAM = 1
MEDICINE_LIST = ["Acetylcysteine", "Apixaban", "..."]

def spam_classifier_heuristic(text: str) -> bool:
    """
    Heuristics to classify whether a string is a spam email or not
    """
    for medicine in MEDICINE_LIST:
        if text.contains(medicine):
            return SPAM
    if text.upper() == text:
        return SPAM
    if re.match(r"([Mm]ake)\s+([\w]+)\s+|\$?\d+\s+((per|for|a)\s+(week|day|year))", text) is not None:
        return SPAM
    else:
        return 1 - SPAM
```

Can we learn simple heuristics?

To apply any kind of learning algorithms we have seen before we need to convert x into a numerical representation h.

- Given a vocabulary $V_{j=1:M}$, determine h such that: $h_j = 1$ whenever token j from the vocabulary is present in x.
- Each datapoint x is represented by an M dimensional vector of 0's and 1's
- How do we determine the vocabulary?
- Just list and count all the words in all of the documents, and then only keep the top M.
- We have numerical features, so just plug them as input into any 'standard' algorithm (ex: Logistic Regression)

Bag of Words

This simple binary representation is called a **(binary) Bag of Words**. = bBoW

- What is included in the representation h of x?
- What if we care about more than just the presence / abscence of a specific word?
- We could just include the count of each word turning h from a vector of 1's and 0's into a vector of counts.
- What about phrases? "Polyethylene Glicol"?

N-Grams

Instead of words being entries in a vocabulary, use phrases.
An **N-gram** is a contiguous sequence of N tokens from a given text.

Under $N = 1$; also called unigrams $= $ BoW

$$V = (\text{“This”}, \text{“is”}, \text{“a”}, \text{“sentence”})$$

$$\text{“This is a sentence”} = (1, 1, 1, 1)$$

$$\text{“A sentence”} = (0, 0, 1, 1)$$

Under $N = 2$; also called bigrams

$$V = (\text{“This is”}, \text{“is a”}, \text{“a sentence”})$$

$$\text{“This is a sentence”} = (1, 1, 1)$$

$$\text{“A sentence”} = (0, 0, 1)$$

Term Frequency

We can represent the "count" (**Term Frequency**) of a word in many different ways:

- Raw count of times it is present in x: **BoW**
- Binarized count of times it is present in x: **binary BoW**
- Count of times it is present in x divided by number of tokens in x
- Raw count scaled by number of other terms (not count of) in x
- Log of the raw count

(number of unique words)

Term Frequency example

V = ("This", "is", "a", "sentence", "another", "not", "Yet")
x_1 = "This is a sentence. This is another sentence."
x_2 = "This is not a sentence. Yet another not a sentence"

= BoW

Word	**TF**(w, x_1)	**TF**(w, x_2)
"This"	2	1
"is"	2	1
"a"	1	2
"sentence"	2	2
"another"	1	1
"not"	0	2
"Yet"	0	1

Table: Term Frequency (TF) for each document

Can be a raw count or a % of words in a document

Inverse Document Frequency

Just like with term frequency, we can represent the "relative prevalence" (**Inverse Document Frequency**) of a word in many different ways:

Denote $N_j = \sum_{i=1}^{N} I(V_j \in x_i)$ count of datapoints that include j-th word in the vocabulary.

Denote N as the total number of documents

- $$\frac{1}{N_j}$$
- $$\frac{N}{N_j}$$
- $$\log\left(\frac{N}{N_j}\right)$$
- $$\log\left(\frac{N}{N_j + 1}\right)$$

This is a scaling factor for how often words occur across documents

IDF Calculation

Word	**Docs with w**	**DF(w)**	**IDF(w)**
"This"	x_1, x_2	2	$\log_{10}(\frac{2}{2}) = \log_{10}(1) = 0$
"is"	x_1, x_2	2	$\log_{10}(\frac{2}{2}) = \log_{10}(1) = 0$
"a"	x_1, x_2	2	$\log_{10}(\frac{2}{2}) = \log_{10}(1) = 0$
"sentence"	x_1, x_2	2	$\log_{10}(\frac{2}{2}) = \log_{10}(1) = 0$
"another"	x_1, x_2	2	$\log_{10}(\frac{2}{2}) = \log_{10}(1) = 0$
"not"	x_2	1	$\log_{10}(\frac{2}{1}) = \log_{10}(2) \approx 0.301$
"Yet"	x_2	1	$\log_{10}(\frac{2}{1}) = \log_{10}(2) \approx 0.301$

Table: Document Frequency (DF) and Inverse Document Frequency (IDF)

V = ("This", "is", "a", "sentence", "another", "not", "Yet")
x_1 = "This is a sentence. This is another sentence."
x_2 = "This is not a sentence. Yet another not a sentence"

TF-IDF

By combining Term Frequency with Inverse Document Frequency we can measure how common the word is in a particular datapoint relative to other documents.

$$\text{TF-IDF}(x) = TF(x) \times IDF(x)$$

Word	**TF**(w, x_2)	**IDF**(w)	**TF-IDF**(w, x_2)
"This"	1	0	$1 \times 0 = 0$
"is"	1	0	$1 \times 0 = 0$
"a"	2	0	$2 \times 0 = 0$
"sentence"	2	0	$2 \times 0 = 0$
"another"	1	0	$1 \times 0 = 0$
"not"	2	0.301	≈ 0.602
"Yet"	1	0.301	≈ 0.301

Table: TF-IDF for Document x_2

Embeddings

All of the methods we talked about can be used to generate numerical representations of whole documents, by the use of just the word occurences, and simple functions. We say that h_i is the **embedding** of x_i, and we call $g(x) = h$ an embedding function. We can then re-frame our problem of learning $f_\theta(x) = y$ as:

$$f_\theta(x) = c_{\theta_1}(h) = c_{\theta_1}(g_{\theta_2}(x)) = y$$

Where c_θ is any classification / regression function, and g_θ is an embedding function.

- Embeddings are not restricted to documents.
- How could we embed an image?
- What if our datapoint consists of an image and text?

Why are embeddings useful?

- A function that can map text, images or other information into the same space is useful
- Compare similarity in a latent space
- Allow us to search, clustering, semi-supervised learning, etc

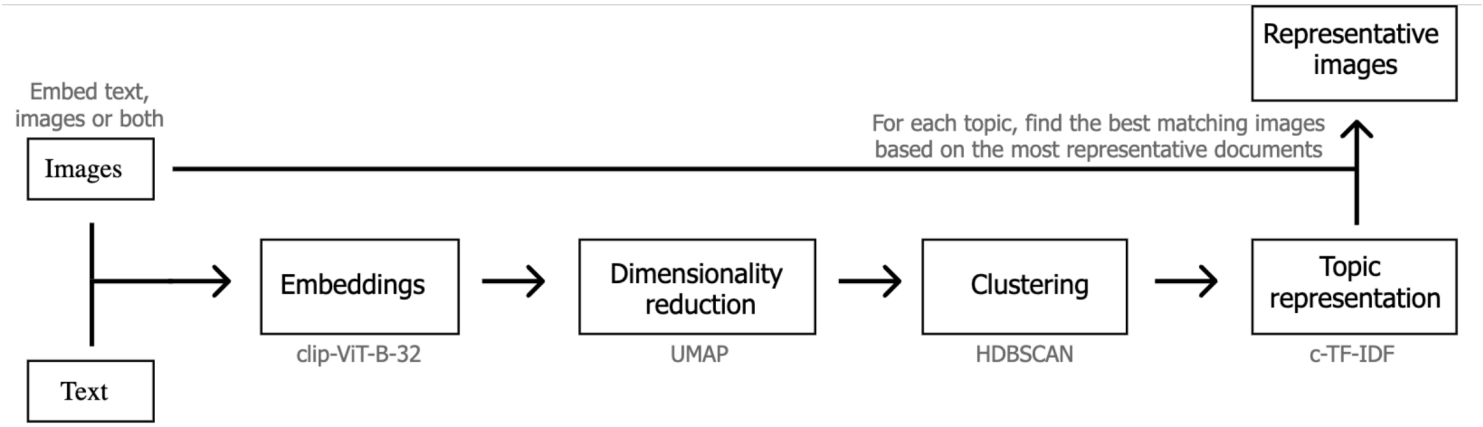


Figure: Multi Modal Clustering - Bert Topic - (Grootendorst, 2022)

We start with a fairly strong assumption:

"Words that have similar meanings will occur in similar contexts"

Based on that we define a **context** of size **k** of token $x_{i,j}$ [1]as a set of tokens:

$$context(x_{i,j}) = \{x_{i,j-k}, x_{i,j-(k-1)}, \ldots, x_{i,j-1}, x_{i,j+1}, \ldots x_{i,j+k}\}$$

Then given a set of datapoints $x_{i=1:N,j=1:M_i}$ and a vocabulary $V_{r=1:M}$ we define an unsupervised learning task of predicting what words occur in the context of each word in the vocabulary. More formally, given a sequence of training words $x_1, \ldots, x_T$ we want to maximize the average log probability:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{j\in context(t)} \log p_\theta(w_j|w_t)$$

[1]This is also called a **skip-gram**

Skip Gram Continued

The basic formulation of $p(x|w)$ uses the softmax function:

$$p_\theta(x|w) = \frac{\exp\left((u(w))^T(v(x))\right)}{\sum_{r=1}^{M}\exp\left((u(w))^T(v(V_r))\right)}$$

where $u(w)$ is the "word" and $v(w)$ is the "context" representation of word w i.e the bBOW.

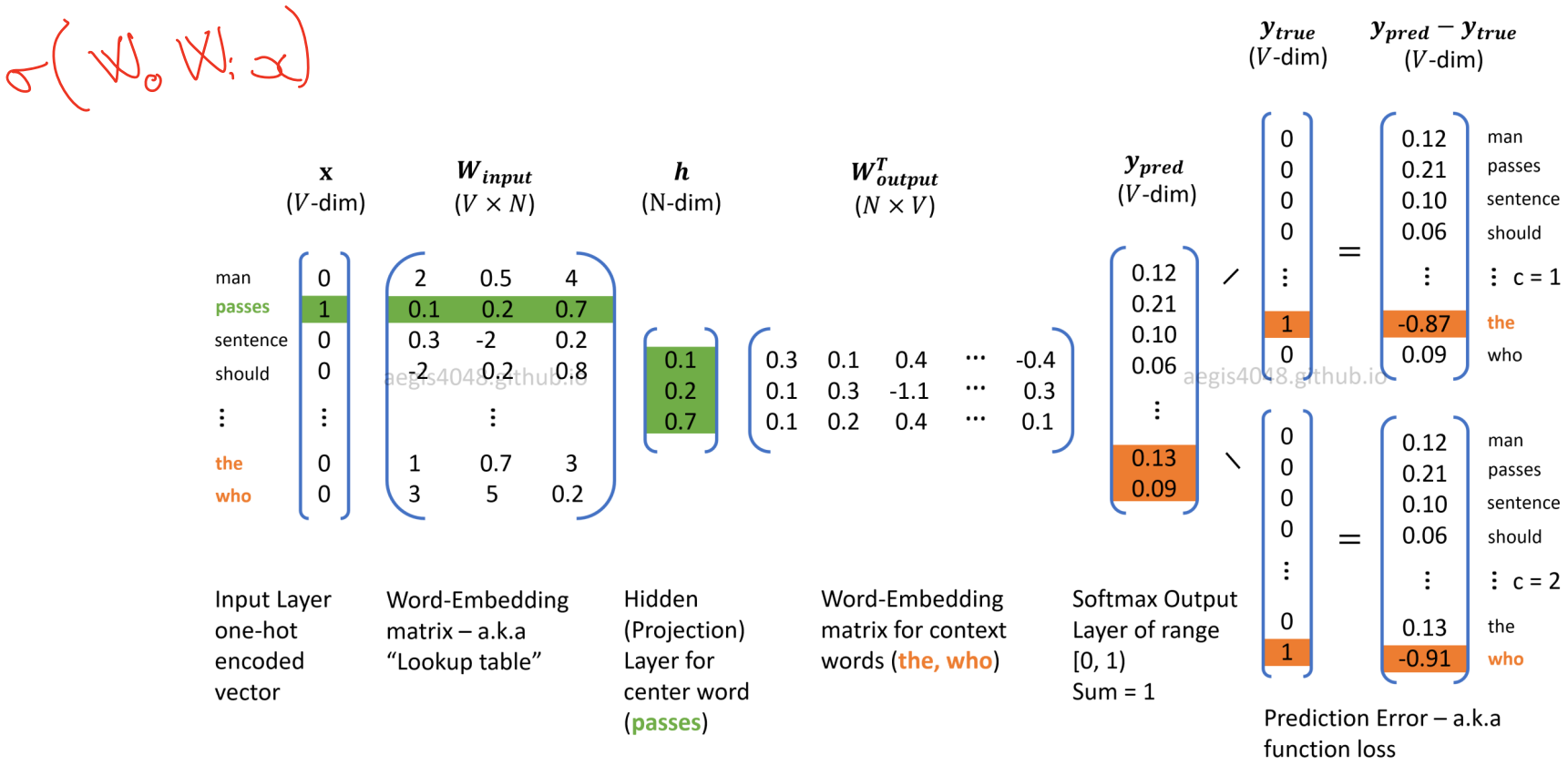
- In our particular case we will take u and v to be simple linear projections of the **one-hot** (binary BoW) encoding of the word, and context respectively.

$$u(w) = U \cdot bBoW(w)$$

The matrix U will be of size $e \times M$ where e is the embedding dimension, which is a hyperparameter you chose.

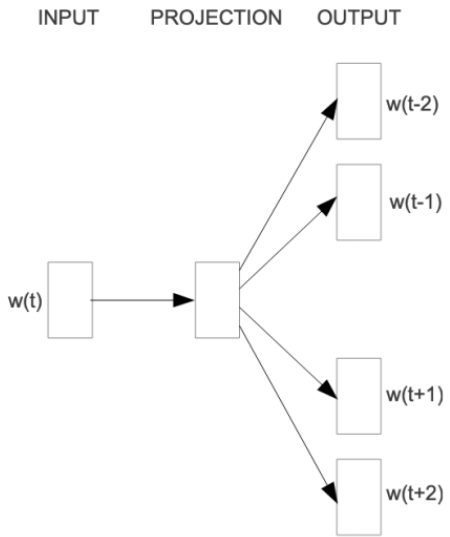
Skip Gram continued

Thinking about this visually we take our word representation, multiply it by the embedding matrix U, get a representation $u(w)$, multiply by U' and get our predicted context words. A nice visual below with some different notation.



Skip Gram Continued

Notice that the bBoW(w) is a binary vector with all 0's and a single 1 at the index of word w in the Vocabulary!
Similarly we define $v(w)$ to be a linear projection that back from $u(w)$ to predict each word in the context.

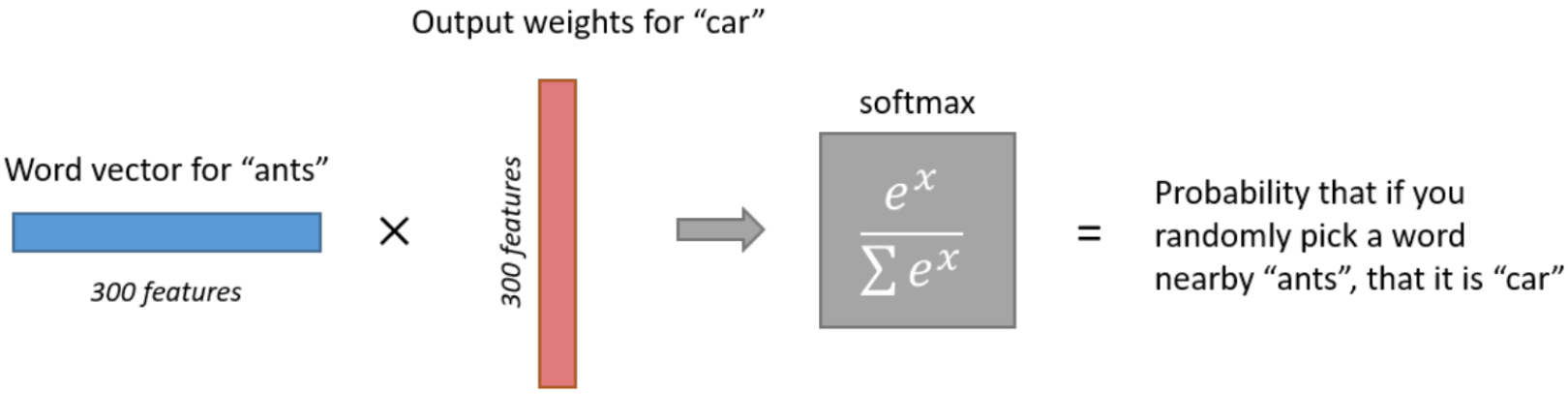


Skip-gram

[1] "Efficient Estimation of Word Representations in Vector Space", Mikolov et al

Skip Gram Vis

How to estimate $p(\text{“car”}|\text{“ants”})$?



[1] Image from: http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

The training process is then as follows:

- Initialize the two matrices
- Sample pairs of words (w_i, w_j) and compute the objective
- Gradient Descent based on the objective.
- After convergence keep only the matrix U
- Embedding of word at vocabulary index i is just the i-th row of U

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = \begin{bmatrix} 10 & 12 & 19 \end{bmatrix}$$

embedding carrying info from tokens

Word2Vec notes

- In practice this training procedure is not feasible - we would have to compute softmax over the entire vocabulary at every step.
- There are a lot of tricks and improvements over the years - really worth reading the original paper.
- There is another possible objective called **Continuous Bag of Words (CBoW)** that is the exact opposite of the Skip Gram Objective - estimate the word given context instead of context given word.

Token classification

Sometimes we care about something more granular than just the whole document. Perhaps we want to identify each parts of text that correspond to certain concepts:

When Sebastian Thrun PERSON started working on self-driving cars at Google ORG in 2007 DATE, few people outside of the company took him seriously. "I can tell you very senior CEOs of major American NORP car companies would shake my hand and turn away because I wasn't worth talking to," said Thrun GPE, now the co-founder and CEO of online higher education startup Udacity, in an interview with Recode earlier this week DATE.

- What should we get a representation of?
- How could we do this?

[1] from SpaCy: https://explosion.ai/demos/displacy-ent

Token Classification and Embeddings continued

- To classify tokens, we can just take the Word2Vec embedding of each token as an input to e.g. Linear Regression / Multinomial Naive Bayes, and estimating probabilities that the token belongs to a certain category.
- We can combine Word2Vec representations into document level representations.
- We can combine Different embedding methods! Nothing is stopping you from taking TF-IDF vector of a document and "stacking" the average of all Word2Vec vectors in the document.

$$\begin{pmatrix} \text{TF-IDF} \\ \frac{1}{N}\sum_{i \in D} u(i) \end{pmatrix}$$

Sequence 2 Sequence Tasks

- What if our output should also be in the form of text?

  (context) words

  - Re-frame the "context" to only feature words before the input
  - Train in the unsupervised setting of CBoW, with the modified context.
  - Idea: Sample the next word, conditional on the previous k based on the CBoW softmax.

Similarity

- What does it mean for 2 words to be similar?
- What does it mean for 2 datapoints to be similar?
- The most common way to measure similarity in NLP is via the cosine similarity

We define **cosine similarity** between two vectors to be:

$$\text{cosine sim}(x, y) = \frac{x \cdot y}{||x|| \cdot ||y||} = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2}\sqrt{\sum_i y_i^2}}$$

This is especially convenient for binary BoW.

Some Cool properties of W2V

Let $h(x)$ be the Word2Vec embedding of the word x. We can perform some vector algebra to find that:

- $h(\text{“Athens”}) - h(\text{“Greece”}) + h(\text{“Germany”}) \cong h(\text{“Berlin”})$
- $h(\text{“Mice”}) - h(\text{“Mouse”}) + h(\text{“Dollar”}) \cong h(\text{“Dollars”})$

In the original paper, they evaluate accuracy on these kinds of “algebraic” operations to find an accuracy of $\sim 55 - 60\%$

You can just download the original Word2Vec embeddings and play around!

Modern tokenizers

- Tokenizers want to encode information into a useful way for LLMs
- Most LLMs have custom trained tokenizers, trying to make training efficient
- LLaMa 3 and DeepSeekv3 - 128k vocab size, GPT-4o - 200k
- Can also tokenize images

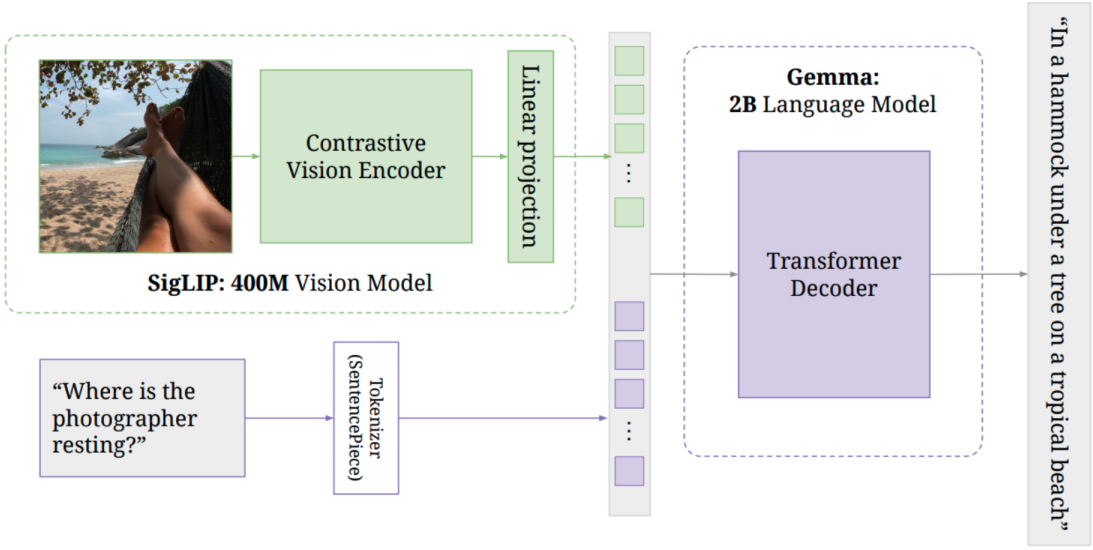


Figure: Paligemma Model - (Beyer et al, 2024)

Summary

- The first modelling step with any data should be to convert it to a numerical representation.
- We can learn embeddings from unlabelled data.
- We can easily combine different kinds of embeddings to improve how we represent data.
- We have learned a number of different document and token embedding algorithms.
- Human Language is hard!

Part 2

- Neural Network Building Blocks
  - Residual Layers
  - Recurrent Layers
  - **Attention**
- Neural Networks
  - Recurrent
  - **Transformer**
- Transformer
  - Encoder
  - Decoder
  - Positional Encoding
  - Attention improvements

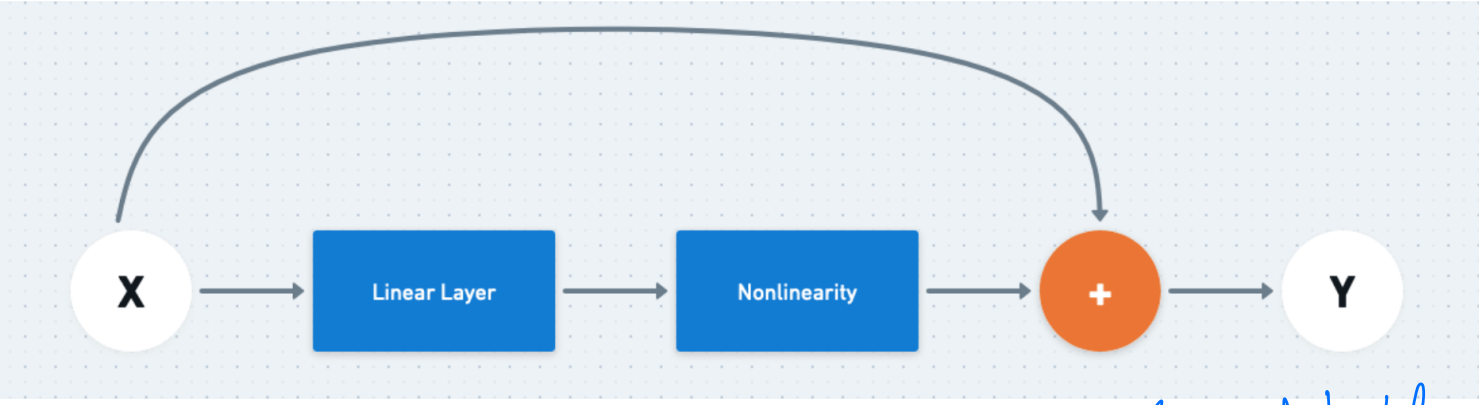
$W^T \dots W^1 x$

$a^T x$

Building Blocks of Neural Networks

If we begin stacking large number of layers together, the signal may get squashed to zero, or blow up to infinity. Similar problem often happens during the gradient computation back through the graph.

To reduce the effect of those problems we often propagate the signal to layers further downstream, in what are called **residual connections**



→ f predicts the residual y - x, easier!

$$y = f(x;\theta) = \phi(Wx + b) + x$$

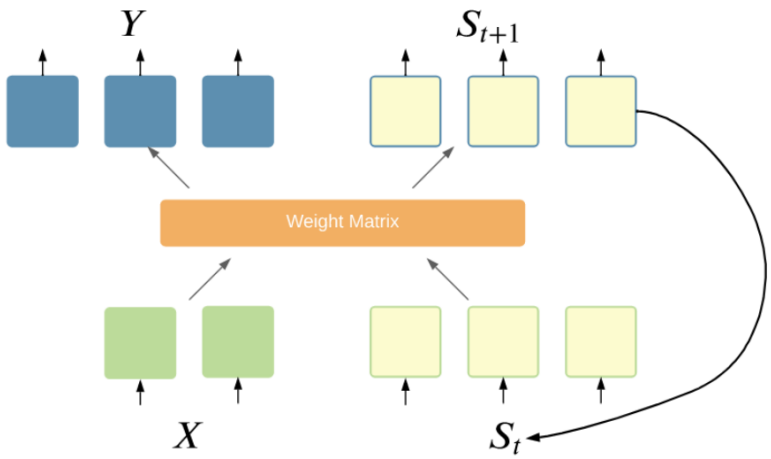
Building Blocks of Neural Networks

When it comes to modelling sequential data (e.g. text, time series), it is often useful to make the model stateful in order for it to help "carry" the information through the graph. To do that we simply add a state at timepoint t: s_t, and computing the output and the new state using some function:

does not depend on t
I am in class

$$(y, s_{t+1}) = f(x, s_t)$$

This is then called a **recurrent layer**.



If we use recurrent layers in our neural network, the outcome is what we typically call a **Recurrent Neural Network** (of which there are many variants). In the simplest possible option the function $f(x, s)$ is a simple Feed-Forward Neural Network. When training RNNs each item in a sequence is used as input, however during inference each item in the sequence will depend on previous predictions.

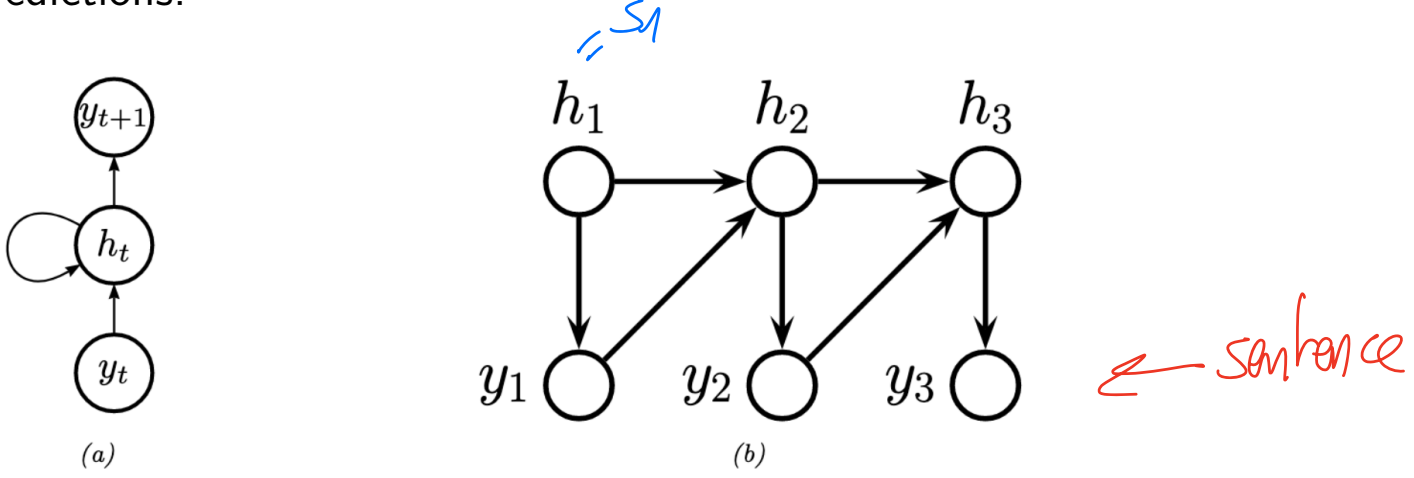


Figure 16.12: Illustration of a recurrent neural network (RNN). (a) With self-loop. (b) Unrolled in time.

Attention is all you need

What if instead of getting just the previous hidden state we were able to take a look at a lot of the previous inputs at once? We could combine all the previous hidden states.

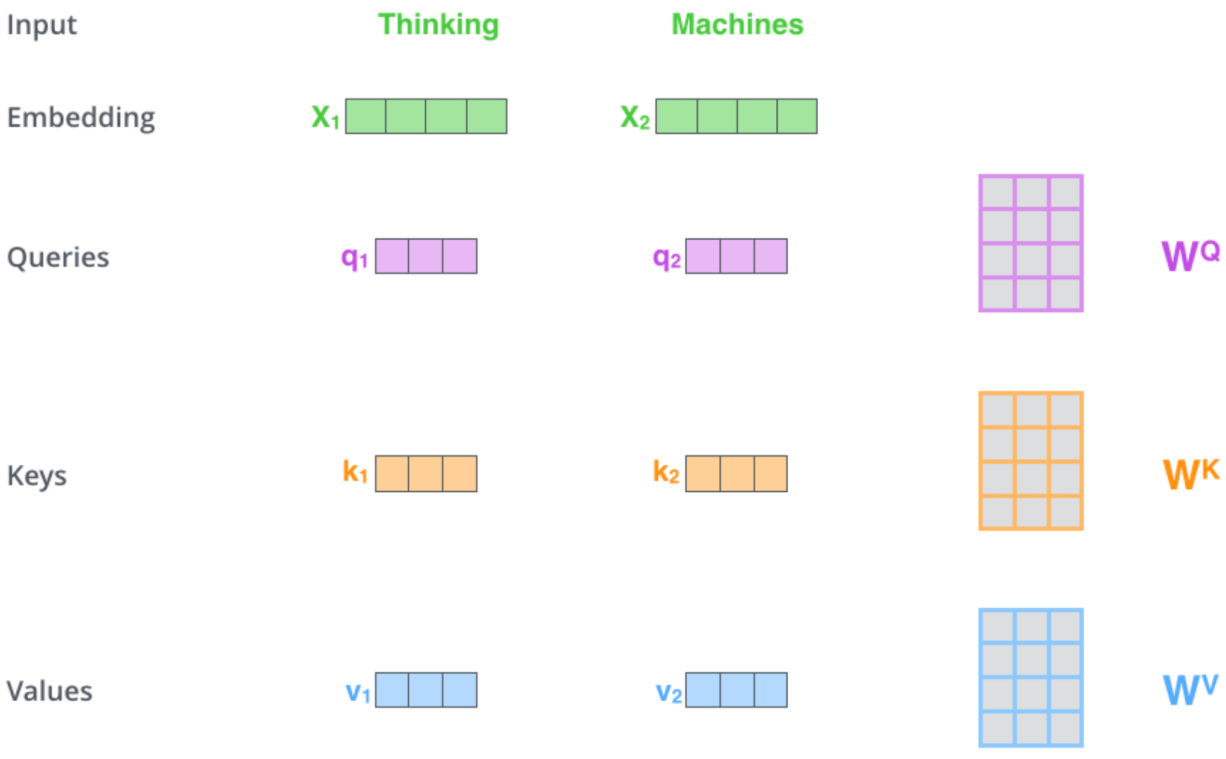
But can we do better? We can score each of the hidden states by how well it is associated with the state we will be predicting.

At a high level the attention mechanism consists of 3 simple steps:

1. Generate a score for each of the hidden states
2. Apply the softmax function to the scores
3. Multiply each of the hidden states by the output of the softmax and add them together.

Attention is all you need

- We can create hidden states by learning different representations
- Create the Query, Key and Value Matricies
- Comes from an information retrieval background



Attention is all you need

⚠ The hard problem remains: how do we score each of the hidden states?
We will begin by creating 3 separate embeddings from each of our inputs, by simply multiplying them by (learned) matrices:

$$q = W^Q x$$
$$k = W^K x$$
$$v = W^V x$$

We then define the **Attention Layer** as:

$$Attn(q, k, v) = \sum_{i=1}^{m} \alpha_i(q, k_i) v_i$$

where α is the scoring function.

(Dot product) Attention is all you need

$$Attn(q, k, v) = \sum_{i=1}^{m} \alpha_i(q, k_i) v_i$$

The most common choice of the attention function is called the **dot product attention**. We obtain the scores by a normalized dot product of the k and q vectors.

$$b(q, k) = \frac{q^T k}{\sqrt{d}}$$

where d is a normalizing constant, usually the dimensionality of the vectors. We then set our attention weights α_i to be the softmax of all the scores:

$$\alpha_i(q, k_i) = \frac{exp(b(q, k_i))}{\sum_{j=1}^{m} exp(b(q, k_j))}$$

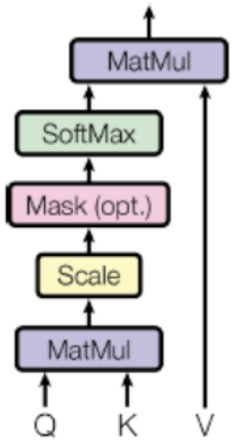
(Dot product) Attention is all you need

The entire process then reduces to:

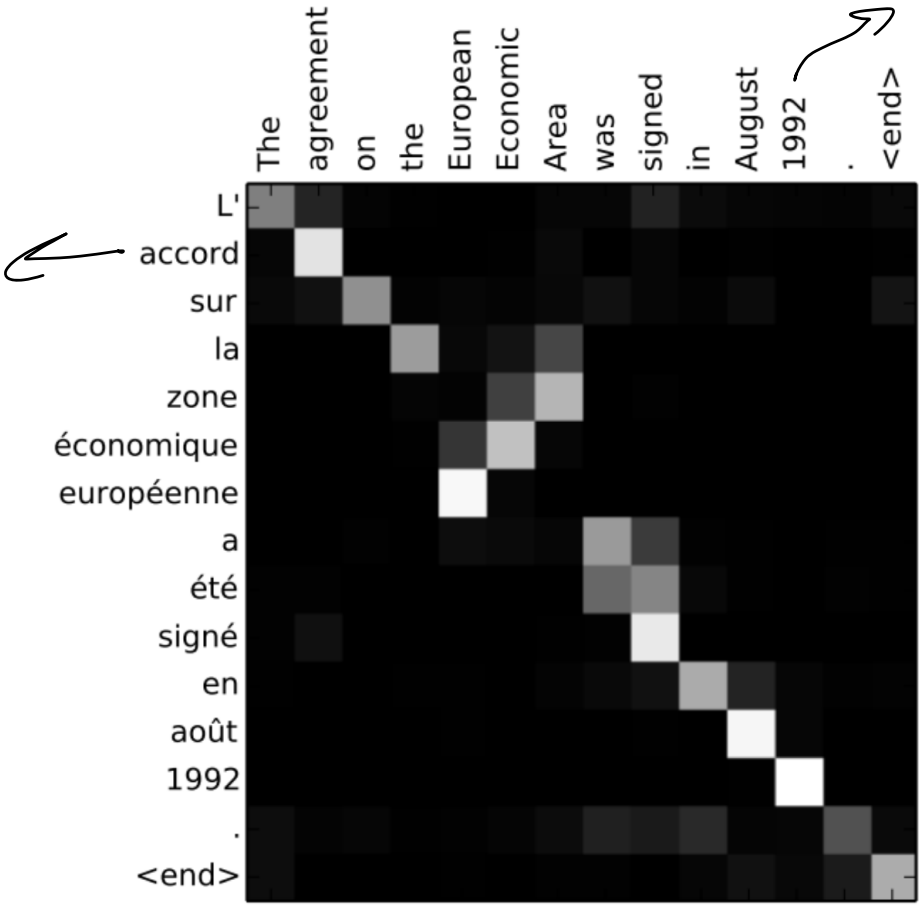
$$Y = Attn(Q, K, V) = \sigma\left(\frac{QK^T}{\sqrt{d}}\right)V$$
$$= \sigma\left(\frac{W^Q X (W^K X)^T}{\sqrt{d}}\right) W^V X$$

$X = (word1, word2, \ldots, word(N-1), word N)$

Scaled Dot-Product Attention



Attention Visualization



query embedding

Keys and Values

For each word, we have a key embed. k_i and a value embedding v_i.

Attention: Bridging Neural Nets and Kernel Methods

From Fixed Weights to Multiplicative Interactions

- Standard neural networks use hidden activations that are linear combinations of inputs with *fixed* weights: $\mathbf{Z} = \varphi(\mathbf{XW})$.
- Attention introduces a more flexible model where the weights depend dynamically on the inputs: $\mathbf{Z} = \varphi(\mathbf{XW}(\mathbf{X}))$, known as a multiplicative interaction.

The Non-parametric Kernel Connection

- Recall kernel-based prediction (like Gaussian Processes). We compare an input query $\boldsymbol{x}$ to training examples $\boldsymbol{x}_i$ using a kernel to obtain similarity scores $\boldsymbol{\alpha} = [\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}_i)]_{i=1}^n$.
- The predicted output (like the GP posterior expectation) is simply a weighted combination of target values: $\hat{\boldsymbol{y}} = \sum_{i=1}^n \alpha_i \boldsymbol{y}_i$.

Attention

- We replace the fixed training examples with learned embeddings $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$.
- The fixed kernel is replaced by a soft attention score, computing outputs in parallel:

$$\mathbf{Z} = \text{Attn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{QK}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

Multi Head Attention and Self Attention

In practice it is advantageous to have multiple "attention heads" each with a different set of W^Q, W^K, W^V matrices.

- Why do you think that is?
- Do we really need all of them?

We then simply concatenate the outputs of all of the attention heads together and multiplied by one final matrix W^O that is learned as well, this is called **Multi Head Attention**.

$$\begin{aligned} o = MHA(Q, K, V) &= Concat(h_1, \ldots, h_h)W^O \\ &= Concat(Attn(Q_1, K_1, V_1), \ldots, Attn(Q_h, K_h, V_h))W^O \end{aligned}$$

Additionally, we can stack several identical Attention / MHA blocks on top each other.

MHA Illustration

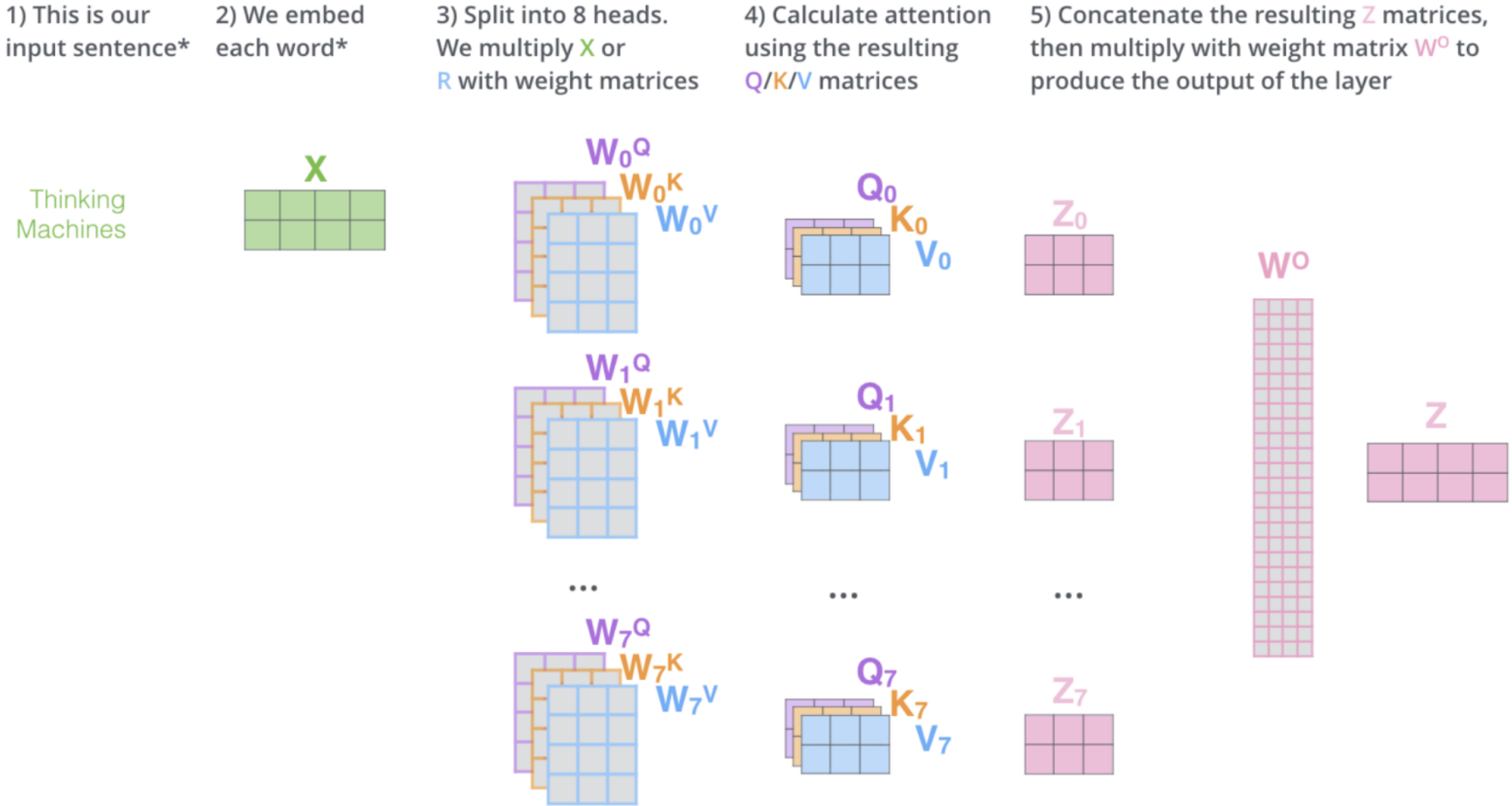
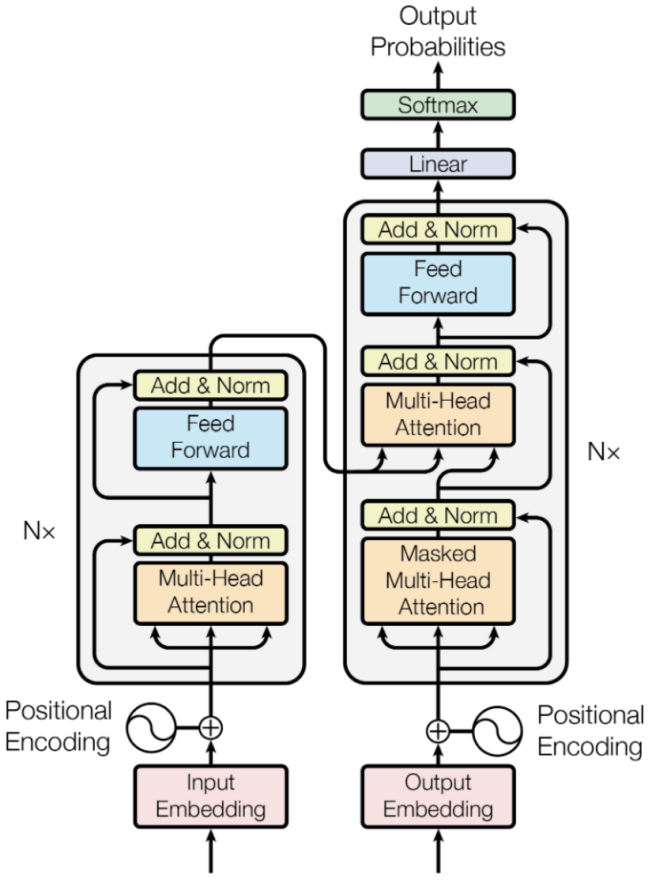


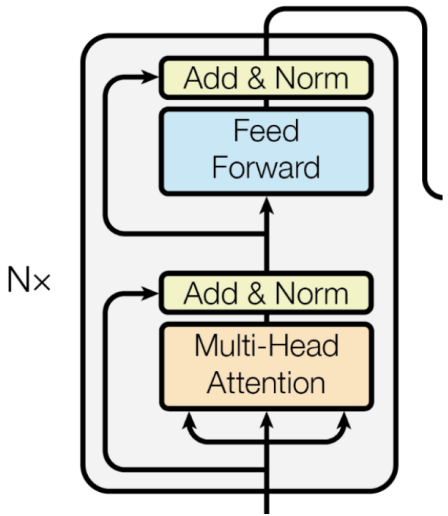
Figure: Illustrated Transformer - (Alammar, 2018)

Transformer

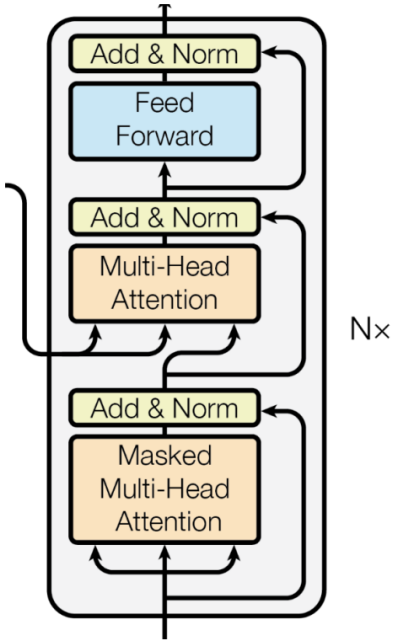
First proposed in a 2017 paper "Attention is all you need", the **Transformer** architecture consists of two stacks (called **Encoder** and **Decoder**) of blocks:



- The **Encoder** consists of a stack of 6 blocks. Each block is further split into two distinct sub-blocks.
- The first is a Multi Head Self Attention mechanism, and the second is a simple FFNN. Both of the sub-blocks have a residual connection around them (followed by normalization).



- Similarily, the **Decoder** is also a stack of 6 blocks.
- However in addition to the two sub-blocks of the encoder, it features a 3rd sub-block.
- This 3rd sub-block performs multi-head attention over the output of the encoder. This "encoder-decoder attention" layer uses Q from the previous decoder layer, and K, V from the output of the encoder.

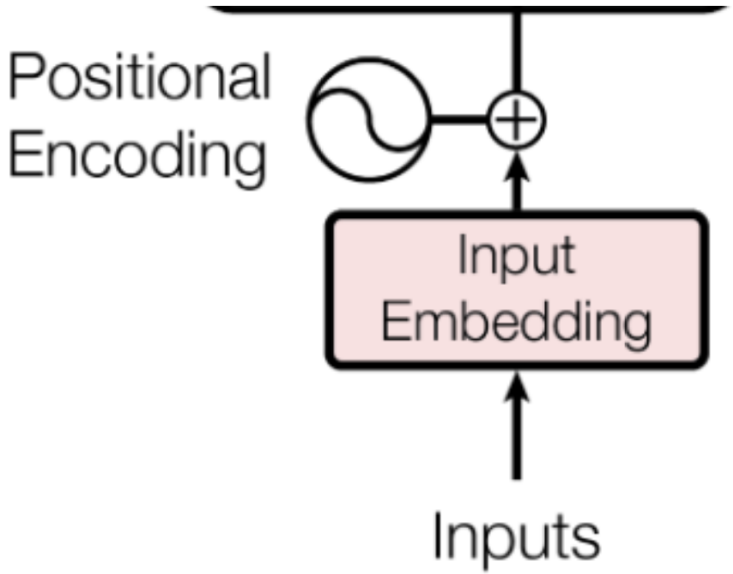


What about inputs?

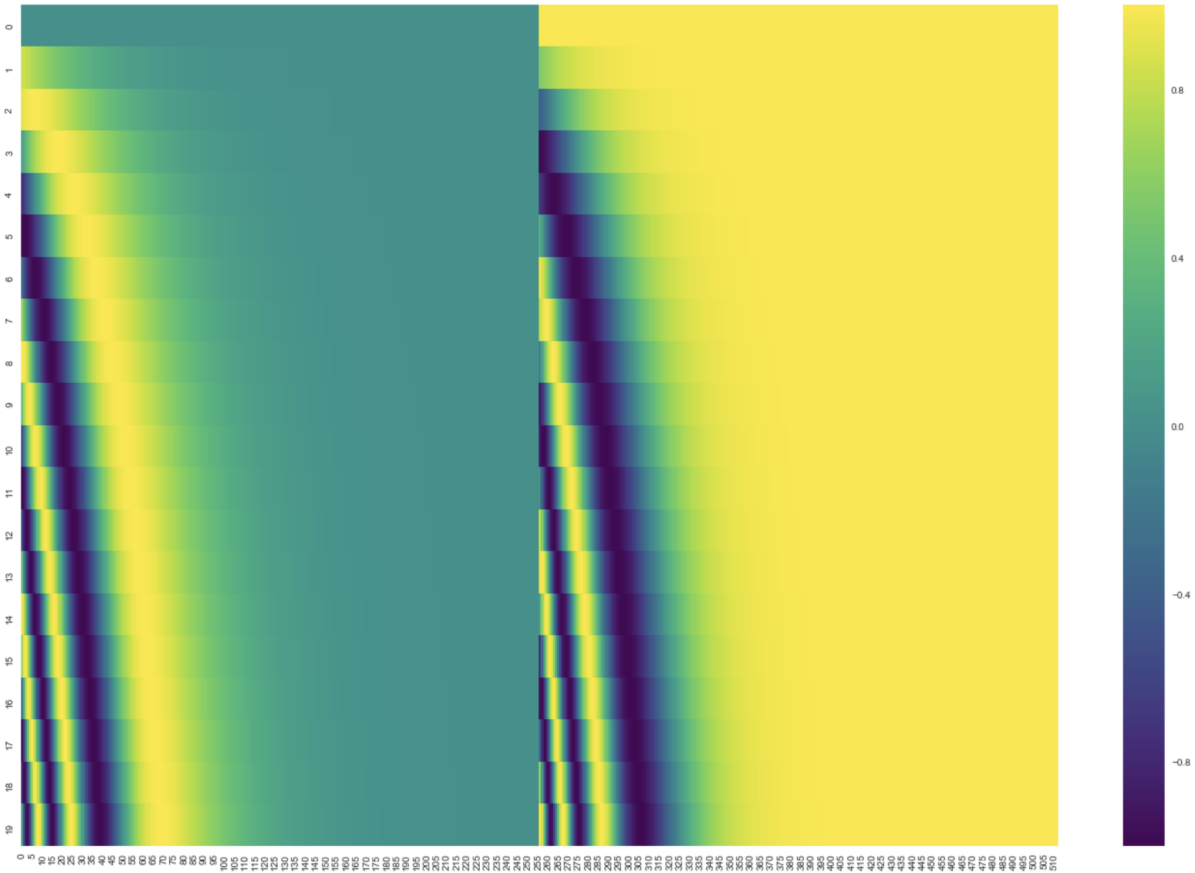
- The input embedding is a learnable "static" **token** embedding similar to the Word2Vec model we have seen above
- Modern GPT embeddings have dimension $O\left(10^3\right)$

What is "Positional Encoding?"

- It's either a learnable embedding (representing position in a sequence), or a predefined embedding.

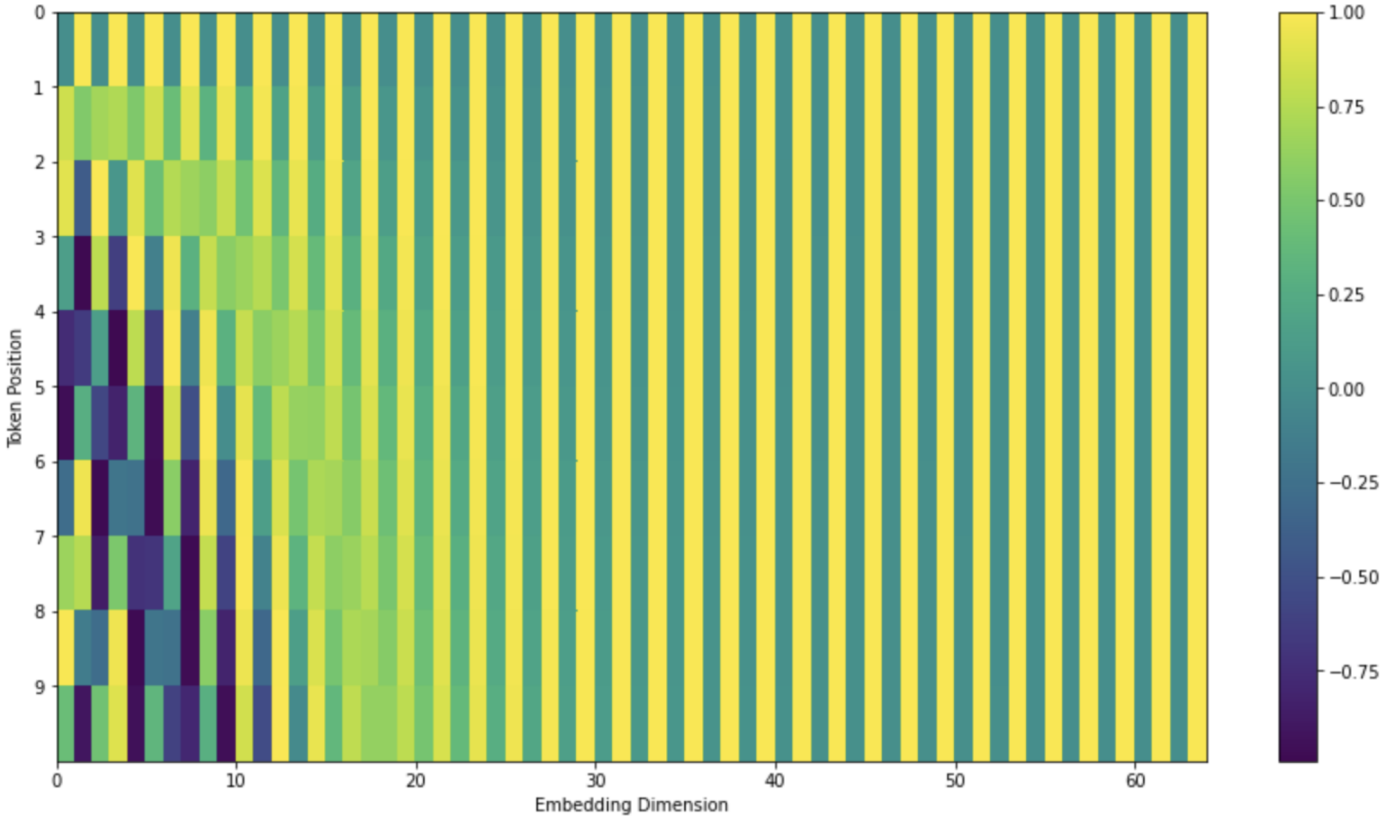


Positional Encoding



A real example of positional encoding for 20 words (rows) with an embedding size of 512 (columns).

Positional Encoding



How to train a Transformer

- The original Transformer model was trained on English ↔ German translations, where at each step the final decoder state was fed into a simple Linear Layer followed by a softmax to produce probabilities over next tokens.
- Currently there are a large number of pre-training tasks (similar in idea to W2V). One of the most common ones is **Masked Language Modelling**, where we randomly replace 15% of tokens with "[MASK]", and the goal of the model is to predict back the original token. BERT (Devlin, 2018) used this approach
- The other approach is to predict the next token based on past tokens only. GPT uses this architecture (Radford, 2018)

Given an unsupervised corpus of tokens $\mathcal{U} = \{u_1, \ldots, u_n\}$, maximize the following likelihood:

$$\mathcal{L}(\mathcal{U}) = \sum_i \log p_\theta(u_i | u_{i-k}, \ldots, u_{i-1})$$

where k is the size of the context window, and the conditional probability p_θ is modeled using a neural network with parameters θ. Use multi-headed self-attention operation over the input context tokens followed by position-wise feedforward layers to produce an output distribution over target tokens:

$$\begin{aligned} h_0 &= UW_e + W_p \\ h_l &= \text{transformer\_block}(h_{l-1}) \quad \forall l \in [1, n] \\ P(u) &= \text{softmax}(h_n W_e^T) \end{aligned}$$

where $U = (u_{-k}, \ldots, u_{-1})$ is the context vector of tokens, n is the number of layers, W_e is the token embedding matrix, and W_p is the position embedding matrix.

Modern Attention

- Attention is expensive because it is quadratic in length
- Linear attention - ex Sliding Windows

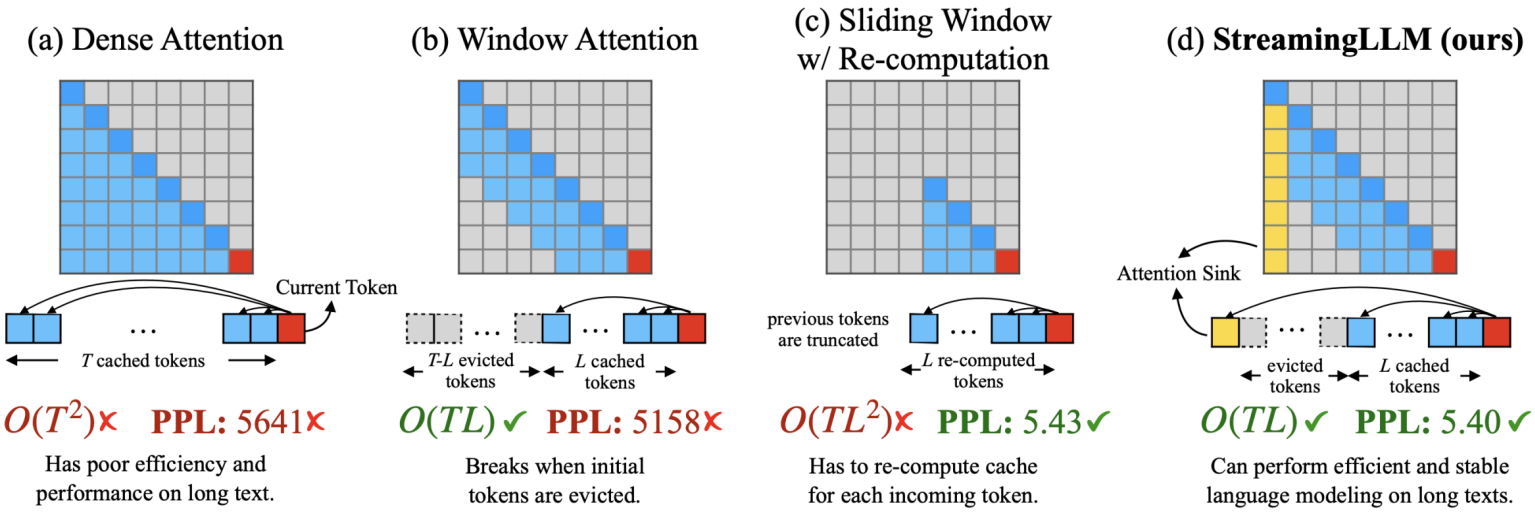


Figure: Attention Sink Example - (Xiao, 2023)

Other questions to think about

- Different types of attention, tradeoffs
- Embedding vector size, precision and dimension
- Different forms of LLM training
- Model training vs inference differences